Towards a Framework for Managing Inconsistencies in Systems of Systems

Conference or Workshop Item

Towards a Framework for Managing Inconsistencies in Systems of Systems

Thiago Viana, Arosha K. Bandara, Andrea Zisman
School of Computing and Communications
The Open University, Walton Hall, Milton Keynes MK7 6AA, UK
{firstname.lastname@open.ac.uk}

ABSTRACT
The growth in the complexity of software systems has led to a proliferation of systems that have been created independently to provide specific functions, such as activity tracking, household energy management or personal nutrition assistance. The runtime composition of these individual systems into Systems of Systems (SoSs) enables support for more sophisticated functionality that cannot be provided by individual constituent systems on their own. However, in order to realize the benefits of these functionalities it is necessary to address a number of challenges associated with SoSs, including, but not limited to, operational and managerial independence, geographic distribution of participating systems, evolutionary development, and emergent conflicting behavior that can occur due interactions between the requirements of the participating systems. In this paper, we present a framework for conflict management in SoSs. The management of conflicting requirements involves four steps, namely (a) overlap detection, (b) conflict identification, (c) conflict diagnosis, and (d) conflict resolution based on the use of a utility function. The framework uses a Monitor-Analyze-Plan-Execute- Knowledge (MAPE-K) architectural pattern. In order to illustrate the work, we use an example SoS ecosystem designed to support food security at different levels of granularity.

CCS Concepts
• Software system structures→Software architectures
• Software functional properties→Correctness→Consistency.

Keywords
Systems of systems; Inconsistency management; Adaptation

1. INTRODUCTION
Software systems have evolved from being stand-alone systems to being composed into Systems of Systems. A System of Systems (SoS) is defined as an arrangement of independently created, discovered, and selected systems, which are integrated into a single system in order to deliver unique capabilities [1]. In this context, each participating system can operate and support different goals in its own environment (viz. *local goals*), as well as support new goals of the SoS as a whole (viz. *global goals*), that could not be achieved separately by the participating systems. An SoS presents many features including, but not limited to operational and managerial independence, geographic distribution of participating systems, and emergent behaviours [2].

In recent years, we have experienced the situation in which individual systems are being composed into bigger systems as SoSs that are capable of delivering unique functionality that spans more complex operating environments. Examples of SoSs with such capabilities are found in transport network systems, household energy management systems, personal nutritional systems, smart homes, smart cities, and intelligent healthcare systems. This evolution of SoSs raises a number of software engineering challenges regarding their specification, design, construction, and operation. Among these challenges, one important challenge is concerned with managing emerging conflicting behaviour expressed as requirements. In an SoS, the various participating systems are often from different domains; are developed by different teams of people under different circumstances and time; have distinct functionalities; and are used by different stakeholders. All of the above factors contribute to the existence of conflicting requirements.

In this paper we present a framework for Managing Conflicting Requirements in Systems of Systems (*MaCoRe_SoS)*. The framework supports four steps for conflict management, namely (a) overlap detection, (b) conflict identification, (c) conflict diagnosis, and (d) conflict resolution. The conflict identification, diagnosis and resolution steps are executed based on a Monitor-Analyze-Plan-Execute- Knowledge (MAPE-K) architectural pattern [3]. The framework assumes requirements specified in an extension of the RELAX language [4]. The overlap detection is executed based on the use of ontologies and identifies requirements that share common elements such as shared resources. The identification of conflicts is assisted by an event monitor component that detects violations of requirements. The diagnosis of the conflicts is performed by an analyzer component using requirements interaction features. The resolution of conflicts is based on the use of a utility function and supports eight resolution methods, namely relaxation, refinement, abandonment, compromise, restructuring, reinforcement, re-planning, and postponement [5].

An example of a food security SoS called *FeedMe FeedMe* [6] is used to illustrate the approach. More specifically, *FeedMe FeedMe* consists of an exemplar of an IoT-based ecosystem developed by some of the authors of this paper to support challenges in food security at four levels of abstractions: individuals, groups, cities, and nations.

The remainder of this paper is structured as follows. In Section 2 we describe *FeedMe FeedMe* SoS example. In Section 3 we present the *MaCoRe-SoS* framework. In Section 4 we discuss related work. Finally, in Section 5 we provide conclusions and future work.

2. Motivating Example – Feed Me Feed Me
In order to illustrate the proposed framework, the approach uses *FeedMe FeedMe* [6], an exemplar of an SoS scenario composed of different computing systems to address food security problems at different levels of granularity (individual, group, city and nation). At the individual level, *FeedMe FeedMe* presents smart devices to monitor, analyse, and provide suggestions about the nutritional and health status of a person. At the group level, *FeedMe FeedMe* uses smart home appliances that interoperate to create a more precise family meal plan, based on the family resources and budget. At the city level, local markets collect data

from multiples families to manage their stock and to reduce food wastage. At the national level, food producers and manufacturers collect data from different markets to forecast food needs and provide alternatives in case of food crisis.

In order to support the granularity levels described above, *FeedMe FeedMe* SoS is composed of four different participating systems: *AnalyseMe*, *HomeHub*, *SmartCity*, and *SmartNation*. Figure 1 shows an overview of the *FeedMe FeedMe* SoS with its various participating systems and devices. We provide below a brief description of the various participating systems.

• **AnalyseMe**. It is a system composed of wearable devices that tracks the health information of a user (e.g., heart rate, blood pressure, blood glucose, food intake, sleep and activity levels), and proposes meal plans together with exercises.

• **HomeHub**. It is a smart system that communicates with the smart appliances in a house aided by smart packaging applications used by refrigerator and pantry devices. This system allows family meals to be planned in advance and to send a list of ingredients to supermarkets when they are required and missing from the house. The HomeHub system acts as a mediator between different smart devices in a house and local supermarkets.

• **SmartCity**. This system supports supermarkets to collect information about communities' grocery requirements and to better manage stocks and inventory.

• **SmartNation**. This system aggregates requirements of individuals, family households, supermarkets, producers, manufacturers, and distributors in order to support management of food production processes.

Conflicts in *FeedMe FeedMe* SoS exist when requirements of the participating systems or requirements of the overall SoS cannot be satisfied due to incorrect use of resources associated with these requirements. For example, consider a requirement of *AnalyseMe* system in which meal plans that satisfies nutritional needs of the user should be created. Consider also the fact that healthy meals are usually more expensive and consume more from home resources such as food, budget, and electricity. Suppose another requirement of *HomeHub* system in which home resources should be consumed as little as possible. In this case, these two requirements may conflict since they are making use of the same resource. Another example is related to the fact that in order to create an accurate family meal plan, the SoS needs to have updated information about the home resources by requesting an hourly update about the consumption of these resources from *HomeHub*. The hourly update about resource consumption may use more electricity and, therefore, it conflicts with the *HomeHub* requirement to use as little electricity as possible.

3. The MaCoRe_SoS Framework

The main goal of the MaCoRe_SoS framework is to manage conflicting requirements in SoSs. The framework supports requirements conflict management based on four steps, as described in the survey from Spanoudakis and Zisman [7] on inconsistency management, namely (i) overlap detection, (ii) conflict identification, (iii) conflict analysis, and (iv) conflict resolution. In the framework, the requirements represent both local goals of the participating components and global goals of the SoS environment as a whole. We distinguish these as local and global requirements.

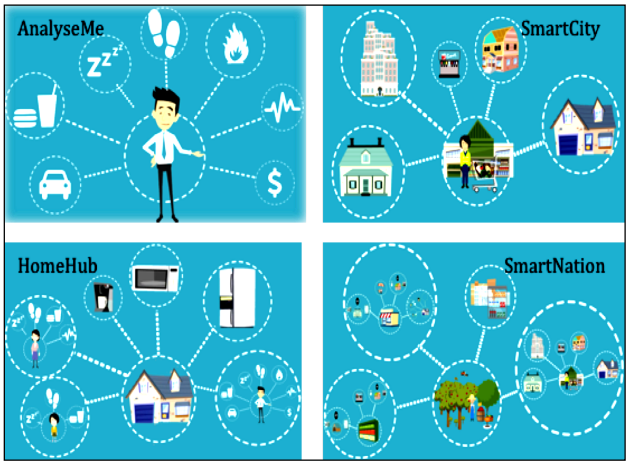

Figure 1. Overview of the FeedMe FeedMe SoS

Figure 2 shows an overview of the framework, illustrating that it supports SoSs environments composed of other stand-alone component systems (CSs), services, or even other systems of systems (SoSs). For simplicity, we will refer to a participating component system, service, or SoS, as an entity.

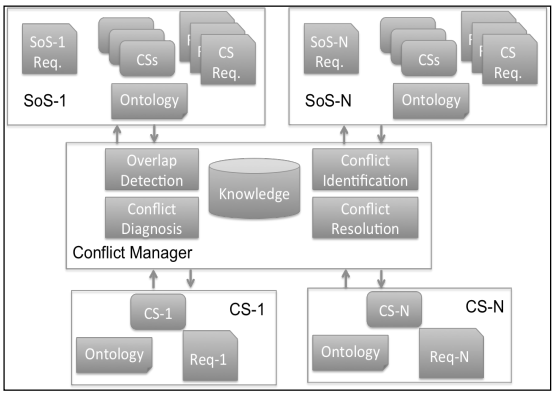

Figure 2. MaCoRe_SoS Framework Overview

For each participant entity, the framework assumes the existence of an ontology and of requirement specifications. The ontology represents concepts of the domain associated with an entity, while a requirements specification represents the requirements of an entity. In the framework, the ontologies are represented in OWL [8] and are used to assist with the identification of elements that are shared by the various participating entities during overlap detection, such as resources, users, or functions.

Systems of systems operate in dynamic environments where the satisfaction of requirements depends on runtime states that are uncertain at design time. In order to accommodate this uncertainty, the requirements in the MaCoRe_SoS framework are specified using an extension of the RELAX language [4], which has a specific support for uncertainty in systems environments.

The conflict identification, diagnosis, and resolution steps in the framework are executed at runtime and based on the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) architectural pattern [3]. The framework includes a database that stores necessary knowledge used during conflict management. The knowledge database contains information about the various requirements, the assertions and events associated with the

requirements, historical data about resolution strategies used in previous conflicts, and information about requirements violations.

3.1 Requirements representation

We suggest to use an extension of the RELAX language [4] to represent the various requirements. RELAX has been proposed as a requirements specification language to represent requirements of self-adaptive systems. RELAX is based on the use of fuzzy logic and is able to address the notion of uncertainty. Similarly, uncertainty is also a characteristic of SoS given the emergent behaviors present in these systems, and the fact that the various entities composing an SoS are developed independently. Another characteristic of RELAX is its flexibility to represent different value ranges, which is necessary when dealing with SoSs with various participating entities using resources in different ways.

The vocabulary used by RELAX is based on a set of modal (e.g., SHALL, MAY … OR), temporal (e.g., EVENTUALLY, UNTIL, AS CLOSE AS POSSIBLE TO), and ordinal (e.g., AS MANY, FEW AS POSSIBLE) operators; as well as uncertainty factors (e.g., ENV, MON, REL, DEP). A full explanation of RELAX can be found in [4].

In order to illustrate our use of RELAX, consider below a local requirement of the *HomeHub* system (HH_R1), and a global requirement (FMFM_R1) of the *FeedMe FeedMe* SoS.

HH_R1 – HomeHub SHALL control the home electricity usage to be AS CLOSE AS POSSIBLE to 100 KWh.

RESOURCE: ELECTRICITY–PROTECT

EVENT: HomeHub-SaveEnergy

FMFM_R1 – The SoS SHALL record data from each family AS EARLY AS POSSIBLE AFTER the day begins and AS CLOSE AS POSSIBLE TO one hour interval thereafter. EVENTUALLY, the SoS SHALL have a synchronized information of all instances of *AnalyseMe* and *HomeHub*.

RESOURCES: ELECTRICITY–CONSUME

EVENT: FMFM-RequestData

The clause AS CLOSE AS POSSIBLE in requirement HH_R1 allows flexible representation of the electricity usage since it can be either above or below 100 KWh. Similarly, in the case of requirement FMFM_R1, the clauses AS CLOSE AS POSSIBLE, AS EARLY AS POSSIBLE AFTER, and EVENTUALLY also support flexibility in the range of resource values. The clause AS EARLY AS POSSIBLE AFTER assists with the situation in which the associated resource is not available, and the system needs to wait until the resource becomes available. The clause EVENTUALLY assists with the situation of conflicts generated by an entity that stops its activities to wait for some resource that is not available at that moment.

In order to support the overlap detection and the conflict identification steps, we have extended the RELAX language with clauses to represent (a) shared elements (such as resources), and (b) events, which are directly related to an action performed and reported by an entity registered into the framework.

As shown in HH_R1 and FMFM_R1, the RELAX specification has been extended with the RESOURCE and EVENT clauses. The RESOURCE clause represents the type of resource associated with the requirement, and how this resource should be used. The different types of resources depend on the domains of the participating systems and the SoS as a whole. For instance, in the case of a smart home nutrition management SoS, relevant resources would include the calorific content of meals or the daily calories consumed by an individual; the quantity and cost of ingredients in meals; energy consumption to prepare meals; and individuals' insulin, cholesterol or blood pressure levels.

A resource can be consumed or protected. In the case of being consumed (CONSUME), the associated value of the resource is decreased. In the case of being protected (PROTECT), the consumption of the associated value of the resource should be prevented. The RESOURCE clause can accommodate the representation of more than one type of resource associated with the requirement.

The EVENT clause represents the different types of events that will trigger the associated requirement. It is possible to have the same event associated with different requirements, and a requirement triggered by different types of events. The event in HH_R1 is concerned with the energy consumption of the various appliances in the *HomeHub* system, while the event in FMFM_R1 is related to data requested by the SoS environment in order to be able to prepare meals for the family. Examples of other types of requirements represented in RELAX for *FeedMe FeedMe* SoS can be found in (http://sead1.open.ac.uk/macore_sos/).

3.2 Overlap Detection / Conflict Identification

As described in [7], the detection of overlaps should be done before indentifying conflicts in order to identify requirements that share common aspects. This is an important activity during conflict management since requirements without overlapping elements cannot be considered as conflicting requirements [9].

In the framework, the overlap detection identifies requirements that share the same elements, such as resources in an SoS. For example, this can be done by using elements described in the RESOURCE clauses of the requirements and the ontologies associated with the various entities. The intersections between requirements and resources of the various participating entities in an SoS provide the possible sources of conflicting requirements, which is stored in the knowledge database for future reference. The same strategy can be used to others shared elements such as users or functions. An example of overlapping requirements is found in HH_R1 and FMFM_R1 presented above. In this case, there is a potential for the requirements to be in conflict since both requirements are concerned with resource "electricity". The MaCoRe_SoS framework executes the identification of conflicts based on the use of assertions represented as fuzzy branching temporal logic (FBTL) [10]. These assertions are generated from the requirements described in the RELAX language which can be mapped to FBTL [4]. As an example the assertion for requirements HH_R1 is presented below.

RELAX Grammar Expression: SHALL (AS CLOSE AS POSSIBLE TO 100 q);

Formal FBTL expression: $AGF((\Delta(q) - 100) \in S)$

Definitions: q is "*HomeHub* control the home electricity usage"; S is a fuzzy set whose membership function has value 1 at zero ($m(0) = 1$) and it decreases continuously around zero; AGF are FBTL quantifiers.

The identification of conflicts is executed by monitoring assertions and detecting violation in the requirements. The framework uses an event monitor component to monitor events and detect requirements violations. When a requirement violation is detected due to a specific event, the framework executes the conflict diagnosis step.

In order to illustrate consider requirement HH_R1 with the assertion presented above and requirement FMFM_R1. Suppose events HomeHub-SaveEnergy and FMFM-RequestData events received by the framework. Assume that FMFM-RequestData requests data from *AnalyseMe* and *HomeHub* systems in the SoS every hour (as per the requirement), causing a higher consumption of electricity. Suppose that the event monitor component detects a violation in the resource electricity of HH_R1 since the value of the home electricity usage is above the expected value (> 100 KWh). In this case, a conflict in HH_R1 is identified, due to resource violation caused by another requirement that uses the same resource.

3.3 Conflict Diagnosis

The conflict diagnosis step analyses a detected violated requirement to verify if a conflict has occurred and to identify the cause of the conflict. As proposed by Robinson et al. [5], the diagnosis is executed based on a set of requirements interaction features, namely: *Basis*, *Degree*, *Direction*, and *Likelihood*.

The Basis feature is concerned with all the identified conflicting requirements and its elements (for instance, the involved resources); the Degree feature represents the requirements satisfaction, in this case this element is represented by the difference (percentage distance) between the actual and the expected value for the specified requirements (for instance, the expected and the actual value of a resource utilization); the Direction feature is calculated based on the Degree feature and can be positive or negative, depending on the satisfaction of the requirement (for instance, if a resource is being over or under used); the Likelihood feature, when available in the framework knowledge database, is based on historical data of past conflict resolution associated with a requirement and a specific resolution strategy. For each requirement and each available resolution method, the framework stores information about the likelihood of solving a conflict using such method in the knowledge database. The framework uses the analyzer component to calculate the above requirements interaction features.

As an example, consider requirement HH_R1 with the actual value for resource electricity as 120 KWh. In this case, the Basis feature is the resource electricity. As the expected value to electricity is 100KWh, the Degree feature is 20%; the Direction feature is positive (the electricity usage is higher than the expected value); and the Likelihood is calculated based on the historical data from each resolution method available in the framework.

The framework supports eight different types of resolution methods: Relaxation, Refinement, Abandonment, Compromise, Postponement, Restructuring, Reenforcement and Replaning, based in [5] and adapted to tackle the SoS context.

Table 1 presents a brief description of each one of the resolution methods and an example of how the framework applies these resolution methods when dealing with conflicts involving shared resources.

3.4 Conflict Resolution

The conflict resolution is executed based on the requirements interaction features identified in the diagnosis step. It uses planner and executor components to support the resolution step. The resolution is based on the use of a utility function that receives as inputs the Basis, Degree, Direction, and Likelihood values. The planner component verifies which resolution method associated with the conflicting requirements should be selected. The executor component carries out the selected resolution method identified by the planner component.

Table 1. MaCoRe_SoS Resolution Methods [5]

Method	Description
Relaxation	Conflicting requirements are relaxed to expand the range of mutually satisfactory options. In the framework the Relaxation resolution method expands the value of a resource concerned with the conflict in order to prevent another resource to run out of the resource. For example, more consumption of calories and more savings in food resources.
Refinement	Conflicting requirements are decomposed into specialized requirements, some of which can be satisfied. In the framework, this resolution method can focus on part of the resources or parts of a requirement, in a way that these parts will be satisfied. For example, in the case of generate meal plans, the method could preserve the resource insulin, but may ignore the resource calories.
Abandonment	Conflicting requirements can be abandoned. In the framework the Abandonment resolution method discards requirements that try to protect a resource.
Compromise	Given a conflict over a value within a domain of values, compromise resolution method finds another substitute value from that domain. In the framework the Compromise resolution method is implemented by searching for new values of resource utilization that manage the actual conflict.
Postponement	Conflict resolution can be postponed. In complex interactions, many conflicts and requirements are interrelated. In the framework the postponement resolution method is implemented by searching for conflicts involving the same resource and by postponing this conflict and resolving other conflicts.
Restructuring	Restructuring methods attempt to change the conflict context; they alter assumptions and related requirements in addition to the conflicting requirements. In the framework the Restructuring resolution method means changes in the structure of a component, or a replacement of a component, in order to manage a conflict.
Reenforcement	Reenforcement is a restructuring that ensures that a precondition is satisfied. In the framework the Reenforcement resolution method reinforces a requirement that protects a resource in order to improve its availability and tries to manage a conflict.
Replanning	Replanning is the selection of an alternative set of requirements to achieve a subordinate requirement. In the framework the Replanning resolution method evaluates the available alternative plans to a specific requirement, and tries to choose the one that is more suitable to the actual resource utilization.

The use of utility functions have been advocated as a technique to support the selection of the best option during decision-making processes [11]. As outlined in [12], utility functions can be used to support autonomic systems to optimize computational resource usage, as it is the case in SoS. For instance, when dealing with conflicting requirements involving shared resources, the main goal of a utility function is to improve availability of resources.

To illustrate the framework we propose to use the utility function given below, which is calculated for each available resolution method (R(RM)) associated with a violated resource:

$$UF(R(R_M)) = C - P + D - L, \quad \text{where}$$

C: is the number of requirements that consume resource R;
P: is the number of requirements that protect resource R;
D: is the calculated Degree of resource R using its Direction;
L: is the Likelihood value of resolving the conflict by method R_M.

In order to illustrate, suppose a conflict involving requirements HH_R1 and FMFM_R1. Assume resolution method Abandonment associated with requirement HH_R1 and resolution method Refinement associated with requirement FMFM_R1. Consider the number of requirements that consume electricity resource (C=26) and the number of requirements that protect electricity resource (P=2). For requirement HH_R1, suppose that the electricity utilization is 120KWh; the Degree value is 20% since the expected value is 100KWh; the Direction is positive (the electricity usage is higher than the expected value); and that based on the framework's historical database, the Likelihood of resolving this type of conflict by using Abandonment on HH_R1 is 10%. In this case, the utility function for this method is given as UF(HH_R1(Abandonment)) = 26-2+20-10 = 34.

For the case of resolution method Refinement, assume that using this method will cause a decrease in electricity consumption in the house to 97KWh. This is due to the fact that using Refinement resolution method on FMFM_R1, the clause "AS CLOSE AS POSSIBLE TO one hour interval thereafter" will be ignored and, therefore, the request of data about a family will stop (reducing the energy consumption). In this case, the Degree value is calculated as 3%; as the expected value is 100KWh and the actual value is 97KWh; and the Direction is negative (the electricity usage is lower than the expected value). Suppose that, based on the framework's historical data, the Likelihood of resolving conflicts involving the requirements HH_R1 and FMFM_R1 by using Refinement is 20%. The utility function is given by UF(FMFM_R1(Refinement)) = 26-2-3-20 = 1. Given the lower value of the utility function for Refinement method, the planner suggests this method of resolution.

Challenges. The use of a shared ontology is a limitation of the framework. The framework assumes that each new entity that registers itself in the framework provides ontology to be integrated into the framework database. An additional challenge is the lack of tools to monitor FBTL expressions; therefore as part of our ongoing work we are developing those tools to be integrated into the framework. Finally, the utility function needs to consider more complex value distributions resource utilization values, as we recognize that a normal distribution is too simplistic to model resources in realistic SoS scenarios. Refinement of the utility function design is another aspect of our ongoing work to develop the MaCoRe_SoS framwork.

4. Related Work

Requirements inconsistency management in stand-alone software systems has been extensively studied in the literature [5][9][13][14][15]. Boehm and Hoh [14] explain the importance to identify and handle conflicts among requirements and proposes the need of a balanced satisfaction between them. Robinson and Pawlowski [15] use a requirements dialog meta-model to present some techniques to manage conflicts and inconsistencies in the requirements documents. Zisman and Kozlenkov [13] use a UML metamodel to propose a goal-based approach to manage conflicts and inconsistencies on design specifications. Further, Nuseibeh *et al.* [9] proposes a framework which is able to detect and diagnose inconsistencies and manage them by solving inconsistencies immediately, ignoring them, or tolerating them for a while. However, all these approaches are executed during design-time and for stand-alone software systems. When dealing with systems of systems, it is necessary to support requirements conflict management during runtime, and for systems that were not created with the intention of being composed.

Some approaches have been proposed to support managing conflicting requirements at runtime [16][17][18][19][20]. Bencomo *et al*. [16] proposes to use requirements reflexion, in which it is possible to have requirements as runtime objects. Their work shows that requirements reflexion is important to support the adaptation process by allowing software systems to reason, understand, explain and modify requirements at runtime. They also describe some of the associated challenges and suggest the use of autonomic computing to support these challenges. Similarly, the MaCoRe_SoS framework uses MAPE-K to assist with conflict management process.

Feather *et al.* [17] brings a goal-driven architecture and a development process to monitor and to reconcile requirements at runtime. Furthermore, Baresi *et al.* [18] presents FLAGS, a goal model that generalizes KAOS model and brings the requirements to runtime as live entities. Moreover, Kneer and Kamsties [19] propose a metamodel which defines the additional requirements artefacts to generate a runtime requirements monitor. However, the above works were not developed to support SoS environments and do not consider monitoring requirements under these environments.

Silva *et al.* [21] presents an approach to deal with runtime evolution of requirements in adaptive systems. Their approach uses the concept of Awareness Requirements and Evolution Requirements. The former indicates situations that require an evolution in the requirements and the latter describes what to do in these situations. However, those requirements needs to be elicited before-hand, and even thought its is possible to specify those requirements regarding individual goals of component systems, their approach don't tackle the problem of requirements that arises as emergent behaviors from the composition of those systems into an SoS. Another important difference from our approach is that they are concerned with evolution in a broader sense, while our approach tackles the problem of managing conflicting requirements.

Vierhauser *et al*. [22] refers to the challenges of monitoring requirements in SoS environments as: "monitoring at different layers, different levels of granularity, across different systems, different technologies, different speeds, with diversity of system requirements, and the performance of the monitoring solution". They propose ReMinds: an adaptable framework to monitor events at runtime in a SoS. The MaCoRE_SoS framework is more complete and uses a MAPE-K architectural pattern to support not only the identification and monitoring of conflicting requirements, but also the diagnosis and resolution of these conflicts.

Pandey and Garlan [20] proposed a hybrid planning approach to the MAPE-K architectural pattern. Their idea is to combine more than one decision-making approach in order to deal with conflicting requirements of planning quickly and finding an optimal plan. Their work is concerned with these two conflicting requirements, while our framework is concerned with conflicting requirements in general and, more specifically, due to resource utilization.

Robinson and Pawlowski [15] states that requirements that deplete a shared resource are a type of conflicting requirement. This specific type of conflict requirement is the main focus of the MaCoRe_SoS at this moment. Lamsweerde *et al*. [23] address the conflicting requirements problem and presents examples of conflicting requirements in a resource management system. Krauter *et al.* [24] presents a survey under the grid computing area and states the importance of an efficient resource management to new and emergent applications such as SoSs.

Malakuti [25] presents the use of formal modelling and verification to detect unexpected and undesired emergent behavior. The work is illustrated in an SoS for the green computing domain and a conflict between performance and energy consumption. His work shows the importance of the resource utilization in an SoS and how it can bring problems. However, his approach is limited to conflict detection, while the MaCoRe_SoS framework supports diagnosis and resolution of conflicts.

5. Conclusion and Future Work

The growth of software systems complexity has led to systems that compose themselves into bigger systems to achieve more sophisticated functionalities. These systems are often System of Systems (SoS) where the management of emerging conflicting behaviors, expressed as requirements is a challenge. As a new and emergent application, an efficient resource management is an important element inside the SoS environment. Therefore, requirements that deplete a shared resource are a type of conflicting requirement. To address this specific type of conflicting requirement we presented the MaCoRe_SoS framework, with four steps: (a) overlap detection, (b) conflict identification, (c) conflict diagnosis, and (d) conflict resolution based on the use of a utility function.

We presented the framework and illustrate it using *FeedMe FeedMe*, an example SoS ecosystem designed to support food security at different levels of granularity (individuals, families, cities, and nations). The framework is able to identify conflicts and after the identification of these conflicts, diagnose them based on requirements interaction features, as proposed by [5]; and choose and apply one of the eight available resolution methods [5] based on a utility function. Currently, we are implementing and evaluating the framework using the *FeedMe FeedMe* exemplar. We plan to evaluate the work in other SoS domains such as transport network or avionics. We are extending RELAX language to support the representation of requirements associated with other sources of possible conflicts like users and behaviour functionalities...We are also developing new utility functions that include requirements prioritization as input, and that allow the use of different conflict resolution methods in parallel.

REFERENCES

[1] Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering, *Systems Engineering Guide for Systems of Systems*, vol. 1. Washington, DC: ODUSD(A&T)SSE, 2008.

[2] M. W. Maier, ‘Architecting principles for systems-of-systems’, in *INCOSE International Symposium*, 1996.

[3] J. O. Kephart and D. M. Chess, ‘The vision of autonomic computing’, *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel, ‘RELAX: a language to address uncertainty in self-adaptive systems requirement’, *Requir. Eng.*, 2010.

[5] W. N. Robinson, S. D. Pawlowski, and V. Volkov, ‘Requirements interaction management’, *ACM Comput. Surv. CSUR*, vol. 35, no. 2, pp. 132–190, 2003.

[6] A. Bennaceur, C. Mccormick, J. Garc__a Gal_an, C. Perera, A. Smith, et and al, ‘Feed me, Feed me: An Exemplar for Engineering Adaptive Software’, presented at the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Austin, USA, 2016.

[7] G. Spanoudakis and A. Zisman, ‘Inconsistency management in software engineering: Survey and open research issues’, *Handb. Softw. Eng. Knowl. Eng.*, vol. 1, pp. 329–380, 2001.

[8] S. Bechhofer, ‘OWL: Web ontology language’, in *Encyclopedia of Database Systems*, Springer, 2009.

[9] B. Nuseibeh, S. Easterbrook, and A. Russo, ‘Leveraging inconsistency in software development’, *Computer*, vol. 33, no. 4, pp. 24–29, 2000.

[10] S. Moon, K. H. Lee, and D. Lee, ‘Fuzzy branching temporal logic’, *IEEE Trans. Syst. Man Cybern. Part B Cybern.*, vol. 34, no. 2, pp. 1045–1055, 2004.

[11] H. Raiffa, *Decision Analysis: Introductory Lectures on Choices Under Undertainty*. Addison-Wesley, 1968.

[12] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, ‘Utility functions in autonomic systems’, in *Autonomic Computing, 2004. Proceedings. International Conference on*, 2004.

[13] A. Kozlenkov and A. Zisman, ‘Discovering, recording, and handling inconsistencies in software specifications’, *Int. J. Comput. Inf. Sci.*, vol. 5, no. 2, pp. 89–108, 2004.

[14] B. Boehm and H. In, ‘Identifying quality-requirement conflicts’, *IEEE Softw.*, vol. 13, no. 2, p. 25, 1996.

[15] W. N. Robinson and S. D. Pawlowski, ‘Managing requirements inconsistency with development goal monitors’, *Softw. Eng. IEEE Trans. On*, vol. 25, 1999.

[16] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, ‘Requirements Reflection: Requirements As Runtime Entities’, in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, New York, NY, USA, 2010, pp. 199–202.

[17] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard, ‘Reconciling System Requirements and Runtime Behavior’, in *Proceedings of the 9th International Workshop on Software Specification and Design*, Washington, DC, USA, 1998, p. 50–.

[18] L. Baresi, L. Pasquale, and P. Spoletini, ‘Fuzzy Goals for Requirements-Driven Adaptation’, in *2010 18th IEEE International Requirements Engineering Conference*, 2010.

[19] F. Kneer and E. Kamsties, ‘Model-based Generation of a Requirements Monitor.’, in *REFSQ Workshops*, 2015.

[20] A. Pandey and D. Garlan, ‘Hybrid Planning For Self-Adaptation’.

[21] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos, ‘(Requirement) evolution requirements for adaptive systems’, in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2012, pp. 155–164.

[22] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel, ‘ReMinds: A flexible runtime monitoring framework for systems of systems’, *J. Syst. Softw.*, vol. 112, pp. 123–136, 2016.

[23] A. Van Lamsweerde, R. Darimont, and E. Letier, ‘Managing conflicts in goal-driven requirements engineering’, *IEEE Trans. Softw. Eng.*, vol. 24, no. 11, pp. 908–926, 1998.

[24] K. Krauter, R. Buyya, and M. Maheswaran, ‘A taxonomy and survey of grid resource management systems for distributed computing’, *Softw. Pract. Exp.*, vol. 32,, 2002.

[25] S. Malakuti, ‘Detecting emergent interference in integration of multiple self-adaptive systems’, in *Proceedings of the 2014 European Conference on Software Architecture Workshops*, 2014, p. 24.